

## Axiomatic Semantics

## Programs → Theorems. Axiomatic Semantics

- Consists of:
  - A language for making assertions about programs
  - Rules for establishing when assertions hold
- Typical assertions:
  - During the execution, only non-null pointers are dereferenced
  - This program terminates with  $x = 0$
- Partial vs. total correctness assertions
  - Safety vs. liveness properties
  - Usually focus on safety (partial correctness)

## Partial Correctness Assertions

- The assertions we make about programs are of the form:

$$\{A\} c \{B\}$$

with the meaning that:

- Whenever we start the execution of  $c$  in a state that satisfies  $A$ , the program either does not terminate or it terminates in a state that satisfies  $B$
- $A$  is called precondition and  $B$  is called postcondition
- For example:  
 $\{y \leq x\} z := x; z := z + 1 \{y < z\}$   
is a valid assertion
- These are called Hoare triples or Hoare assertions

## Total Correctness Assertions

- $\{A\} c \{B\}$  is a partial correctness assertion. It does not imply termination
- $[A] c [B]$  is a total correctness assertion meaning that  
Whenever we start the execution of  $c$  in a state that satisfies  $A$  the program does terminate in a state that satisfies  $B$
- Now let's be more formal
  - Formalize the language of assertions,  $A$  and  $B$
  - Say when an assertion holds in a state
  - Give rules for deriving Hoare triples

## Languages for Assertions

- A specification language
  - Must be easy to use and expressive (conflicting needs)
    - Most often only expression  $\odot$
  - Syntax: how to construct assertions
  - Semantics: what assertions mean
- Typical examples
  - First-order logic
  - Temporal logic (used in protocol specification, hardware specification)
  - Special-purpose languages: Z, Larch, Java ML

## State-Based Assertions

- Assertions that characterize the state of the execution
  - Recall: state = state of locals + state of memory
- Our assertions will need to be able to refer to
  - Variables
  - Contents of memory
- What are not state-based assertions
  - Variable  $x$  is live, lock  $L$  will be released
  - There is no correlation between the values of  $x$  and  $y$

## An Assertion Language

- We'll use a fragment of first-order logic first
  - Formulas  $P ::= A \mid T \mid \perp \mid P_1 \wedge P_2 \mid \forall x.P \mid P_1 \Rightarrow P_2 \mid$
  - Atoms  $A ::= f(A_1, \dots, A_n) \mid E_1 \leq E_2 \mid E_1 = E_2 \mid \dots$
- We can also have an arbitrary assortment of function symbols
  - $\text{ptr}(E, T)$  - expression  $E$  denotes a pointer to  $T$
  - $E : \text{ptr}(T)$  - same in a different notation
  - $\text{reachable}(E_1, E_2)$  - list cell  $E_2$  is reachable from  $E_1$
  - these can be built-in or defined

Automated Deduction - George Necula - Lecture 2

7

## Semantics of Assertions

- We introduced a language of assertions, we need to assign meanings to assertions.
  - We ignore for now references to memory
- Notation  $\rho, \sigma \models A$  to say that an assertion holds in a given state.
  - This is well-defined when  $\rho$  is defined on all variables occurring in  $A$  and  $\sigma$  is defined on all memory addresses referenced in  $A$
- The  $\models$  judgment is defined inductively on the structure of assertions.

Automated Deduction - George Necula - Lecture 2

8

## Semantics of Assertions

- Formal definition (we drop  $\sigma$  for simplicity):

$\rho \models \text{true}$  always  
 $\rho \models e_1 = e_2$  iff  $\rho \vdash e_1 \Downarrow n_1$  and  $\rho \vdash e_2 \Downarrow n_2$  and  $n_1 = n_2$   
 $\rho \models e_1 \geq e_2$  iff  $\rho \vdash e_1 \Downarrow n_1$  and  $\rho \vdash e_2 \Downarrow n_2$  and  $n_1 \geq n_2$   
 $\rho \models A_1 \wedge A_2$  iff  $\rho \models A_1$  and  $\rho \models A_2$   
 $\rho \models A_1 \vee A_2$  iff  $\rho \models A_1$  or  $\rho \models A_2$   
 $\rho \models A_1 \Rightarrow A_2$  iff  $\rho \models A_1$  implies  $\rho \models A_2$   
 $\rho \models \forall x.A$  iff  $\forall n \in \mathbb{Z}, \rho[x:=n] \models A$   
 $\rho \models \exists x.A$  iff  $\exists n \in \mathbb{Z}, \rho[x:=n] \models A$

Automated Deduction - George Necula - Lecture 2

9

## Semantics of Assertions

- Now we can define formally the meaning of a partial correctness assertion
  - $\models \{A\} c \{B\}$ :  
 $\forall \rho \sigma. \forall \rho' \sigma'. (\rho, \sigma \models A \wedge \rho, \sigma \vdash c \Downarrow \rho', \sigma') \Rightarrow \rho', \sigma' \models B$
- ... and the meaning of a total correctness assertion
  - $\models [A] c [B]$  iff  
 $\forall \rho \sigma. \forall \rho' \sigma'. (\rho, \sigma \models A \wedge \rho, \sigma \vdash c \Downarrow \rho', \sigma') \Rightarrow \rho', \sigma' \models B$   
 $\wedge$   
 $\forall \rho \sigma. \rho, \sigma \models A \Rightarrow \exists \rho' \sigma'. \rho, \sigma \vdash c \Downarrow \rho', \sigma'$

Automated Deduction - George Necula - Lecture 2

10

## Why Isn't This Enough?

- Now we have the formal mechanism to decide when  $\{A\} c \{B\}$ 
  - Start the program in all states that satisfies  $A$
  - Run the program
  - Check that each final state satisfies  $B$
- This is exhaustive testing
- Not enough
  - Can't try the program in all states satisfying the precondition
  - Can't find all final states for non-deterministic programs
  - And also it is impossible to effectively verify the truth of a  $\forall x.A$  postcondition (by using the definition of validity)

Automated Deduction - George Necula - Lecture 2

11

## Derivations as Proxies for Validity

- We define a symbolic technique for deriving valid assertions from others that are known to be valid
  - We start with validity of first-order formulas
- We write  $\vdash A$  when we can derive (prove) the assertion  $A$ 
  - We wish that  $(\forall \rho \sigma. \rho, \sigma \models A)$  iff  $\vdash A$
- We write  $\vdash \{A\} c \{B\}$  when we can derive (prove) the partial correctness assertion
  - We wish that  $\models \{A\} c \{B\}$  iff  $\vdash \{A\} c \{B\}$

Automated Deduction - George Necula - Lecture 2

12

### Derivation Rules for Assertions

- The derivation rules for  $\vdash A$  are the usual ones from first-order logic with
- Natural deduction style axioms:

$$\frac{\vdash A \quad \vdash B}{\vdash A \wedge B} \quad \frac{\vdash [a/x]A \quad (a \text{ is fresh})}{\vdash \forall x.A} \quad \frac{\vdash \forall x.A}{\vdash [E/x]A}$$

$$\frac{\vdash A \quad \dots \quad \vdash B}{\vdash A \Rightarrow B} \quad \frac{\vdash A \Rightarrow B \quad \vdash A}{\vdash B} \quad \frac{\vdash [E/x]A}{\vdash \exists x.A} \quad \frac{\vdash \exists x.A \quad \vdash B}{\vdash B} \quad \frac{\vdash [a/x]A \quad \dots \quad \vdash B}{\vdash B}$$

Automated Deduction - George Necula - Lecture 2

13

### Derivation Rules for Hoare Triples

- Similarly we write  $\vdash \{A\} c \{B\}$  when we can derive the triple using derivation rules
- There is one derivation rule for each command in the language
- Plus, the rule of consequence

$$\frac{\vdash A' \Rightarrow A \quad \vdash \{A\} c \{B\} \quad \vdash B \Rightarrow B'}{\vdash \{A'\} c \{B'\}}$$

Automated Deduction - George Necula - Lecture 2

14

### Derivation Rules for Hoare Logic

- One rule for each syntactic construct:

$$\frac{}{\vdash \{A\} \text{ skip } \{A\}}$$

$$\frac{\vdash \{A\} c_1 \{B\} \quad \vdash \{B\} c_2 \{C\}}{\vdash \{A\} c_1; c_2 \{C\}}$$

$$\frac{\vdash \{A \wedge b\} c_1 \{B\} \quad \vdash \{A \wedge \neg b\} c_2 \{B\}}{\vdash \{A\} \text{ if } b \text{ then } c_1 \text{ else } c_2 \{B\}}$$

Automated Deduction - George Necula - Lecture 2

15

### Derivation Rules for Hoare Logic (II)

- The rule for while is not syntax directed
  - It needs a loop invariant

$$\frac{\vdash \{A \wedge b\} c \{A\}}{\vdash \{A\} \text{ while } b \text{ do } c \{A \wedge \neg b\}}$$

- Exercise: try to see what is wrong if you make changes to the rule (e.g., drop " $\wedge b$ " in the premise, ...)

Automated Deduction - George Necula - Lecture 2

16

### Hoare Rules: Assignment

- Example:  $\{A\} x := x + 2 \{x \geq 5\}$ . What is  $A$ ?
  - $A$  has to imply  $x \geq 3$
- General rule:

$$\frac{}{\vdash \{[e/x]A\} x := e \{A\}}$$

- Surprising how simple the rule is!
- But try  $\{A\} *x = 5 \{ *x + *y = 10 \}$ 
  - $A$  is " $*y = 5$  or  $x = y$ "
  - How come the rule does not work?

Automated Deduction - George Necula - Lecture 2

17

### Example: Assignment

- Assume that  $x$  does not appear in  $e$ 
  - Prove that  $\{\text{true}\} x := e \{x = e\}$
- We have

$$\frac{}{\vdash \{e = e\} x := e \{x = e\}}$$

because  $[e/x](x = e) \equiv e = [e/x]e \equiv e = e$

- Assignment + consequence:

$$\frac{\vdash \text{true} \Rightarrow e = e \quad \vdash \{e = e\} x := e \{x = e\}}{\vdash \{\text{true}\} x := e \{x = e\}}$$

Automated Deduction - George Necula - Lecture 2

18

## The Assignment Axiom (Cont.)

- Hoare said: "Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic."

- Caveats are sometimes needed for languages with aliasing:

- If  $x$  and  $y$  are aliased then
  - { true }  $x := 5$  {  $x + y = 10$  }
  - is true

## Multiple Hoare Rules

- For some constructs multiple rules are possible:

$$\frac{}{\vdash \{A\} x := e \{ \exists x_0. [x_0/x] A \wedge x = [x_0/x] e \}}$$

(This was the "forward" axiom for assignment before Hoare)

$$\frac{\vdash A \wedge b \Rightarrow C \quad \vdash \{C\} c \{A\} \quad \vdash A \wedge \neg b \Rightarrow B}{\vdash \{A\} \text{ while } b \text{ do } c \{B\}}$$

$$\frac{\vdash \{C\} c \{b \Rightarrow C \wedge \neg b \Rightarrow B\}}{\vdash \{b \Rightarrow C \wedge \neg b \Rightarrow B\} \text{ while } b \text{ do } c \{B\}}$$

- Exercise: these rules can be derived from the previous ones using the consequence rules

## Example: Conditional

$$\frac{\begin{array}{l} D_1 :: \vdash \{ \text{true} \wedge y \leq 0 \} x := 1 \{ x > 0 \} \\ D_2 :: \vdash \{ \text{true} \wedge y > 0 \} x := y \{ x > 0 \} \end{array}}{\vdash \{ \text{true} \} \text{ if } y \leq 0 \text{ then } x := 1 \text{ else } x := y \{ x > 0 \}}$$

- $D_1$  is obtained by consequence and assignment

$$\frac{\begin{array}{l} \vdash \{ 1 > 0 \} x := 1 \{ x > 0 \} \\ \vdash \text{true} \wedge y \leq 0 \Rightarrow 1 > 0 \end{array}}{\vdash \{ \text{true} \wedge y \leq 0 \} x := 1 \{ x > 0 \}}$$

- $D_2$  is also obtained by consequence and assignment

$$\frac{\begin{array}{l} \vdash \{ y > 0 \} x := y \{ x > 0 \} \\ \vdash \text{true} \wedge y > 0 \Rightarrow y > 0 \end{array}}{\vdash \{ \text{true} \wedge y > 0 \} x := y \{ x > 0 \}}$$

## Example: Loop

- We want to derive that

$$\vdash \{ x \leq 0 \} \text{ while } x \leq 5 \text{ do } x := x + 1 \{ x = 6 \}$$

- Use the rule for while with invariant  $x \leq 6$

$$\frac{\begin{array}{l} \vdash x \leq 6 \wedge x \leq 5 \Rightarrow x + 1 \leq 6 \quad \vdash \{ x + 1 \leq 6 \} x := x + 1 \{ x \leq 6 \} \\ \vdash \{ x \leq 6 \wedge x \leq 5 \} x := x + 1 \{ x \leq 6 \} \end{array}}{\vdash \{ x \leq 6 \} \text{ while } x \leq 5 \text{ do } x := x + 1 \{ x \leq 6 \wedge x > 5 \}}$$

- Then finish-off with consequence

$$\frac{\begin{array}{l} \vdash x \leq 0 \Rightarrow x \leq 6 \\ \vdash x \leq 6 \wedge x > 5 \Rightarrow x = 6 \quad \vdash \{ x \leq 6 \} \text{ while } \dots \{ x \leq 6 \wedge x > 5 \} \end{array}}{\vdash \{ x \leq 0 \} \text{ while } \dots \{ x = 6 \}}$$

## Another Example

- Verify that

$$\vdash \{A\} \text{ while true do } c \{B\}$$

holds for any  $A$ ,  $B$  and  $c$

- We must construct a derivation tree

$$\frac{\begin{array}{l} \vdash A \Rightarrow \text{true} \quad \vdash \{ \text{true} \wedge \text{true} \} c \{ \text{true} \} \\ \vdash \text{true} \wedge \text{false} \Rightarrow B \quad \vdash \{ \text{true} \} \text{ while true do } c \{ \text{true} \wedge \text{false} \} \end{array}}{\vdash \{A\} \text{ while true do } c \{B\}}$$

- We need an additional lemma:

$$\forall c. \vdash \{ \text{true} \} c \{ \text{true} \}$$

- How do you prove this one?

## GCD Example

- Let  $c$  be the program:

```
while (x ≠ y) do
  if (x ≤ y)
    then y := y - x
    else x := x - y
```

- We'll derive that

$$\vdash \{ x = m \wedge y = n \} c \{ x = \text{gcd}(m, n) \}$$

### GCD Example (2)

- Crucial to select good loop invariant

- Let the precondition  $Pre$  be
 
$$x = m \wedge y = n$$

- Let the postcondition  $Post$  be
 
$$x = \text{gcd}(m, n)$$

We use the loop invariant

$$I \stackrel{\text{def}}{=} \text{gcd}(x, y) = \text{gcd}(m, n)$$

### GCD Example (3)

We first use the rule of consequence to obtain the subgoal

$$\vdash \{I\} c \{I \wedge \neg(x \neq y)\} \quad (1)$$

But we also need to prove

$$\vdash Pre \Rightarrow I \quad (2)$$

$$\vdash I \wedge \neg(x \neq y) \Rightarrow Post \quad (3)$$

Subgoal 2 reduces to

$$x = m \wedge y = n \Rightarrow \text{gcd}(x, y) = \text{gcd}(m, n)$$

Subgoal 3 reduces to

$$\text{gcd}(x, y) = \text{gcd}(m, n) \wedge x = y \Rightarrow x = \text{gcd}(m, n)$$

### GCD Example (4)

Now we still have to derive subgoal 1:

$$\vdash \{I\} c \{I \wedge \neg(x \neq y)\}$$

We can apply the rule for `while` and we get the subgoal

$$\vdash \{I \wedge x \neq y\} d \{I\} \quad (4)$$

where  $d$  is the body of the loop:

```

if(x ≤ y)
  then y := y - x
  else x := x - y
  
```

### GCD Example (5)

We can derive subgoal 4 using the rule for conditionals and we get two subgoals

$$\vdash \{I \wedge x \neq y \wedge x \leq y\} y := y - x \{I\} \quad (5)$$

$$\vdash \{I \wedge x \neq y \wedge x > y\} x := x - y \{I\} \quad (6)$$

Each of the subgoals 5 and 6 can be derived using the rule of consequence followed by assignment:

$$\vdash I \wedge x \neq y \wedge x \leq y \Rightarrow \text{gcd}(m, n) = \text{gcd}(x, y - x) \quad (7)$$

$$\vdash I \wedge x \neq y \wedge x > y \Rightarrow \text{gcd}(m, n) = \text{gcd}(x - y, y) \quad (8)$$

### GCD Example (6)

$$\vdash I \wedge x \neq y \wedge x > y \Rightarrow \text{gcd}(m, n) = \text{gcd}(x - y, y)$$

- The above can be proved by realizing that  $\text{gcd}(x, y) = \text{gcd}(x - y, y)$

- Q.e.d.
- This completes the proof
- We used a lot of arithmetic
- We had to invent the loop invariants

- What about the proof for total correctness?

### Hoare Rule for Function Call

- If no recursion we can inline the function call

$$\frac{f(x_1, \dots, x_n) = C_f \in \text{Program} \{ A \} C_f \{ B[f/x] \}}{\vdash \{ A[e_1/x_1, \dots, e_n/x_n] \} x := f(e_1, \dots, e_n) \{ B \}}$$

- In general,
  1. each function  $f$  has a  $Pre_f$  and  $Post_f$

$$\vdash \{ Pre_f[e_1/x_1, \dots, e_n/x_n] \} x := f(e_1, \dots, e_n) \{ Post_f[x/f] \}$$

2. For each function we check  $\{ Pre_f \} C_f \{ Post_f \}$

## Axiomatic Semantics in Presence of Side-Effects

## Naïve Handling of Program State

- We allow memory read in assertions:  $*x + *y = 5$
- We try:
  - $\{ A \} *x = 5 \{ *x + *y = 10 \}$
- A ought to be " $*y = 5$  or  $x = y$ "
- The Hoare rule would give us:
  - $(*x + *y = 10)[5/*x]$
  - $= 5 + *y = 10$
  - $= *y = 5$  (we lost one case)
- How come the rule does not work?

## Handling Program State

- We cannot have side-effects in assertions
  - While creating the theorem we must remove side-effects!
  - But how to do that when lacking precise aliasing information?
- Important technique: Postpone alias analysis
- Model the state of memory as a symbolic mapping from addresses to values:
  - If  $E$  denotes an address and  $M$  a memory state then:
  - $sel(M, E)$  denotes the contents of memory cell
  - $upd(M, E, V)$  denotes a new memory state obtained from  $M$  by writing  $V$  at address  $E$

## More on Memory

- We allow variables to range over memory states
  - So we can quantify over all possible memory states
- And we use the special pseudo-variable  $\mu$  in assertions to refer to the current state of memory
- Example:

$$" \forall i. i \geq 0 \wedge i < 5 \Rightarrow sel(\mu, A + i) > 0 " = \text{allpositive}(\mu, A, 0, 5)$$

says that entries 0..4 in array  $A$  are positive

## Semantics of Memory Expressions

- We need a new kind of values (memory values)

Values  $v ::= n \mid a \mid \sigma$

$$\frac{}{\rho, \sigma \vdash \mu \Downarrow \sigma}$$

$$\frac{\rho, \sigma \vdash E_m \Downarrow \sigma' \quad \rho, \sigma \vdash E_a \Downarrow a}{\rho, \sigma \vdash sel(E_m, E_a) \Downarrow \sigma'(a)}$$

$$\frac{\rho, \sigma \vdash E_m \Downarrow \sigma' \quad \rho, \sigma \vdash E_a \Downarrow a \quad \rho, \sigma \vdash E_v \Downarrow v}{\rho, \sigma \vdash upd(E_m, E_a, E_v) \Downarrow \sigma'[a := v]}$$

## Hoare Rules: Side-Effects

- To correctly model writes we use memory expressions
  - A memory write changes the value of memory

$$\frac{}{\{ B[upd(\mu, E_1, E_2)/\mu] \} *E_1 := E_2 \{ B \}}$$

- Important technique: treat memory as a whole
- And reason later about memory expressions with inference rules such as (McCarthy):

$$sel(upd(M, E_1, E_2), E_3) = \begin{cases} E_2 & \text{if } E_1 = E_3 \\ sel(M, E_3) & \text{if } E_1 \neq E_3 \end{cases}$$

## Memory Aliasing

- Consider again:  $\{ A \} *x = 5 \{ *x + *y = 10 \}$
- We obtain:
 
$$\begin{aligned}
 A &= (*x + *y = 10)[\text{upd}(\mu, x, 5)/\mu] \\
 &= (\text{sel}(\mu, x) + \text{sel}(\mu, y) = 10) [\text{upd}(\mu, x, 5)/\mu] \\
 &= \text{sel}(\text{upd}(\mu, x, 5), x) + \text{sel}(\text{upd}(\mu, x, 5), y) = 10 \quad (*) \\
 &= 5 + \text{sel}(\text{upd}(\mu, x, 5), y) = 10 \\
 &= \text{if } x = y \text{ then } 5 + 5 = 10 \text{ else } 5 + \text{sel}(\mu, y) = 10 \\
 &= x = y \text{ or } *y = 5 \quad (**)
 \end{aligned}$$
- To (\*) is theorem generation
- From (\*) to (\*\*) is theorem proving

Automated Deduction - George Necula - Lecture 2

37

## Alternative Handling for Memory

- Reasoning about aliasing is expensive (NP-hard)
- Sometimes completeness is sacrificed with the following (approximate) rule:

$$\text{sel}(\text{upd}(M, E_1, E_2), E_3) = \begin{cases} E_2 & \text{if } E_1 = (\text{obviously}) E_3 \\ \text{sel}(M, E_3) & \text{if } E_1 \neq (\text{obviously}) E_3 \\ p & \text{otherwise (p is a fresh new parameter)} \end{cases}$$

- The meaning of "obvious" varies:
  - The addresses of two distinct globals are  $\neq$
  - The address of a global and one of a local are  $\neq$
- PREFIX and GCC use such schemes

Automated Deduction - George Necula - Lecture 2

38

## Using Hoare Rules. Notes

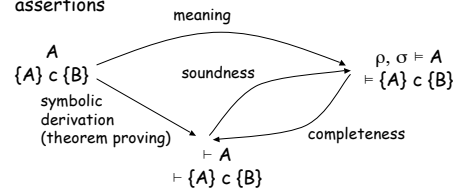
- Hoare rules are mostly syntax directed
- There are three wrinkles:
  - When to apply the rule of consequence?
  - What invariant to use for while?
  - How do you prove the implications involved in consequence?
- The last one is how theorem proving gets in the picture
  - This turns out to be doable!
  - The loop invariants turn out to be the hardest problem! (Should the programmer give them? See Dijkstra.)

Automated Deduction - George Necula - Lecture 2

39

## Where Do We Stand?

- We have a language for asserting properties of programs
- We know when such an assertion is true
- We also have a symbolic method for deriving assertions



Automated Deduction - George Necula - Lecture 2

40

## Soundness of Axiomatic Semantics

- Formal statement
 
$$\text{If } \vdash \{ A \} c \{ B \} \text{ then } \models \{ A \} c \{ B \}$$
- or, equivalently
 
$$\text{For all } \rho, \sigma, \text{ if } \rho, \sigma \models A \text{ and } D :: \rho, \sigma \vdash c \Downarrow \rho', \sigma' \text{ and } H :: \vdash \{ A \} c \{ B \} \text{ then } \rho', \sigma' \models B$$

Automated Deduction - George Necula - Lecture 2

41

## Completeness of Axiomatic Semantics Weakest Preconditions

Automated Deduction - George Necula - Lecture 2

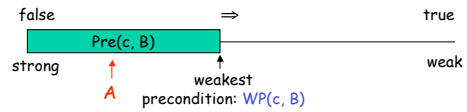
42

### Completeness of Axiomatic Semantics

- Is it true that whenever  $\models \{A\} c \{B\}$  we can also derive  $\vdash \{A\} c \{B\}$  ?
- If it isn't then it means that there are valid properties of programs that we cannot verify with Hoare rules
- Good news: for our language the Hoare triples are complete
- Bad news: only if the underlying logic is complete (whenever  $\models A$  we also have  $\vdash A$ )
  - this is called relative completeness

### Proof Idea

- Dijkstra's idea: To verify that  $\{A\} c \{B\}$ 
  - Find out all predicates  $A'$  such that  $\models \{A'\} c \{B\}$ 
    - call this set  $Pre(c, B)$
  - Verify for one  $A' \in Pre(c, B)$  that  $A \Rightarrow A'$
- Assertions can be ordered:



- Thus: compute  $WP(c, B)$  and prove  $A \Rightarrow WP(c, B)$

### Proof Idea (Cont.)

- Completeness of axiomatic semantics:
  - If  $\models \{A\} c \{B\}$  then  $\vdash \{A\} c \{B\}$
- Assuming that we can compute  $wp(c, B)$  with the following properties:
  - $wp$  is a precondition (according to the Hoare rules)
    - $\vdash \{wp(c, B)\} c \{B\}$
  - $wp$  is the weakest precondition
    - If  $\models \{A\} c \{B\}$  then  $\models A \Rightarrow wp(c, B)$
$$\frac{\vdash A \Rightarrow wp(c, B) \quad \vdash \{wp(c, B)\} c \{B\}}{\vdash \{A\} c \{B\}}$$
- We also need that whenever  $\models A$  then  $\vdash A$  !

### Weakest Preconditions

- Define  $wp(c, B)$  inductively on  $c$ , following Hoare rules:

$$\frac{\{A\} c_1 \{C\} \quad \{C\} c_2 \{B\}}{\{A\} c_1; c_2 \{B\}}$$

$$wp(c_1; c_2, B) = wp(c_1, wp(c_2, B))$$

$$\frac{\{ [e/x]B \}}{\{ [e/x]B \} x := E \{ B \}}$$

$$wp(x := e, B) = [e/x]B$$

$$\frac{\{A_1\} c_1 \{B\} \quad \{A_2\} c_2 \{B\}}{\{ E \Rightarrow A_1 \wedge \neg E \Rightarrow A_2 \} \text{ if } E \text{ then } c_1 \text{ else } c_2 \{ B \}}$$

$$wp(\text{if } E \text{ then } c_1 \text{ else } c_2, B) = E \Rightarrow wp(c_1, B) \wedge \neg E \Rightarrow wp(c_2, B)$$

### Weakest Preconditions for Loops

- We start from the equivalence
  - $\text{while } b \text{ do } c = \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}$
- Let  $w = \text{while } b \text{ do } c$  and  $W = wp(w, B)$
- We have that
  - $W = b \Rightarrow wp(c, W) \wedge \neg b \Rightarrow B$
- But this is a recursive equation !
  - We know how to solve these using domain theory
- We need a domain for assertions

### A Partial-Order for Assertions

- What is the assertion that contains least information?
  - true - does not say anything about the state
- What is an appropriate information ordering?
  - $A \leq A' \text{ iff } \models A' \Rightarrow A$
- Is this partial order complete?
  - Take a chain  $A_1 \leq A_2 \leq \dots$
  - Let  $\bigwedge A_i$  be the infinite conjunction of  $A_i$ 
    - $\sigma \models \bigwedge A_i \text{ iff for all } i \text{ we have that } \sigma \models A_i$
  - Verify that  $\bigwedge A_i$  is the least upper bound
- Can  $\bigwedge A_i$  be expressed in our language of assertions?
  - In many cases yes (see Winskel), we'll assume yes

## Weakest Precondition for WHILE

- Use the fixed-point theorem
  - $F(A) = b \Rightarrow wp(c, A) \wedge \neg b \Rightarrow B$
  - Verify that  $F$  is both monotonic and continuous
- The least-fixed point (i.e. the weakest fixed point) is
  - $wp(w, B) = \bigwedge F^i(\text{true})$

## Weakest Preconditions (Cont.)

- Define a family of wp's
  - $wp_k(\text{while } e \text{ do } c, B)$  = weakest precondition on which the loop if it terminates in  $k$  or fewer iterations, it terminates in  $B$
  - $wp_0 = \neg E \Rightarrow B$
  - $wp_1 = E \Rightarrow wp(c, wp_0) \wedge \neg E \Rightarrow B$
  - ...
- $wp(\text{while } e \text{ do } c, B) = \bigwedge_{k \geq 0} wp_k = \text{lub} \{wp_k \mid k \geq 0\}$
- Is it the case that  $wp_k \Rightarrow wp_{k-1}$ ? The opposite?
- Weakest preconditions are
  - Impossible to compute (in general)
  - Can we find something easier to compute yet sufficient?

## Weakest Precondition. Example 1

- Consider the code:
  - while  $x \leq 5$  do  $x := x + 1$
  - with postcondition  $P = x \geq 7$
- What is the weakest precondition?
- $wp(x := x + 1, A) = A[x+1/x]$
- $WP_0 = \neg(x \leq 5) \Rightarrow x \geq 7 = x \notin \{6\}$
- $WP_1 = x \leq 5 \Rightarrow x + 1 \notin \{6\} \wedge WP_0 = x \notin \{5, 6\}$
- $WP_2 = x \leq 5 \Rightarrow x + 1 \notin \{5, 6\} \wedge WP_0 = x \notin \{4, 5, 6\}$
- ...
- $WP = x \geq 7$

## Weakest Precondition. Example 2

- Consider the code:
  - while  $x \geq 5$  do  $x := x + 1$
  - with postcondition  $P = x \geq 7$
- What is the weakest precondition?
- $wp(x := x + 1, A) = A[x+1/x]$
- $WP_0 = \neg(x \geq 5) \Rightarrow x \geq 7 = x \geq 5$
- $WP_1 = x \geq 5 \Rightarrow x + 1 \geq 7 \wedge WP_0 = x \geq 5$
- ...
- $WP = x \geq 5$

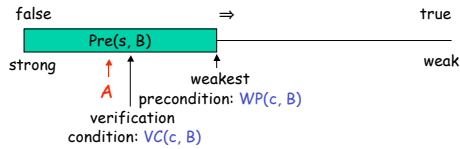
## Theorem Proving and Program Analysis (again)

- Predicates form a lattice:
    - $WP(s, B) = \text{lub}_s(\text{Pre}(s, B))$
  - This is not obvious at all:
    - $\text{lub} \{P_1, P_2\} = P_1 \vee P_2$
    - $\text{lub } PS = \bigvee_{P \in PS} P$
    - But can we always write this with a finite number of  $\vee$ ?
  - Even checking implication can be quite hard
  - Compare with program analysis in which lattices are of finite height and quite simple
- Program Verification is Program Analysis on the lattice of first order formulas

## Verification Conditions

## Not Quite Weakest Preconditions

- Recall what we are trying to do:



- We shall construct a verification condition:  $VC(c, B)$ 
  - The loops are annotated with loop invariants!
  - VC is guaranteed stronger than WP
  - But hopefully still weaker than A:  $A \Rightarrow VC(c, B) \Rightarrow WP(c, B)$

Automated Deduction - George Necula - Lecture 2

55

## Verification Conditions

- Factor out the hard work
  - Loop invariants
  - Function specifications
- Assume programs are annotated with such specs.
  - Good software engineering practice anyway
- We will assume that the new form of the while construct includes an invariant:
  - $\text{while}_I b \text{ do } c$
  - The invariant formula must hold every time before  $b$  is evaluated

Automated Deduction - George Necula - Lecture 2

56

## Invariants Are Not Easy

- Consider the following code from QuickSort

```
int partition(int *a, int L0, int H0, int pivot) {
  int L = L0, H = H0;
  while(L < H) {
    while(a[L] < pivot) L++;
    while(a[H] > pivot) H--;
    if(L < H) { swap a[L] and a[H] }
  }
  return L
}
```

- Consider verifying only memory safety
- What is the loop invariant for the outer loop?

Automated Deduction - George Necula - Lecture 2

57

## Verification Condition Generation (1)

- Mostly follows the definition of the wp function

```
VC(skip, B) = B
VC(c1; c2, B) = VC(c1, VC(c2, B))
VC(if b then c1 else c2, B) = b ⇒ VC(c1, B) & ¬b ⇒ VC(c2, B)
VC(x := e, B) = B[e/x]
VC(let x = e in c, B) = VC(c, B) [e/x] (wrong!)
VC(*e1 := e2, B) = B [upd(μ, e1, e2)/μ]
VC(while b do c, B) = ?
```

Automated Deduction - George Necula - Lecture 2

58

## Verification Condition Generation for WHILE

$$VC(\text{while}_I e \text{ do } c, B) = I \wedge (\forall x_1 \dots x_n. I \Rightarrow (e \Rightarrow VC(c, I) \wedge \neg e \Rightarrow B))$$

$I$  holds on entry       $I$  is preserved in an arbitrary iteration       $B$  holds when the loop terminates in an arbitrary iteration

- $I$  is the loop invariant (provided externally)
- $x_1, \dots, x_n$  are all the variables modified in  $c$
- The  $\forall$  is similar to the  $\forall$  in mathematical induction:  $P(0) \wedge \forall n \in \mathbb{N}. P(n) \Rightarrow P(n+1)$

Automated Deduction - George Necula - Lecture 2

59

## Verification Conditions for Function Calls

- Recall that
  - $\mu$  is the pseudo-variable denoting the memory
  - $f$  is the pseudo-variable used in function  $f$  for return value
- $VC(x := f(e_1, \dots, e_n), B) = \text{Pre}_f[e_i/x_i] \wedge \forall \mu \forall f \text{ Post}_f \wedge B[f/x]$
- We check the precondition
  - with actuals substituted for formals
- We assume that the memory and "f" are changed
- We check that function postcondition is stronger than  $B$

Automated Deduction - George Necula - Lecture 2

60

## Soundness and Completeness of VCGen

- We must prove that, for all  $c$  and  $B$   
 $\models \{ VC(c, B) \} c \{ B \}$
- Or, equivalently  
 $VC(c, B) \Rightarrow WP(c, B)$
- Try it (the case for loops is interesting)
- Completeness is trickier
- One formulation: it is possible to choose the loop invariants such that  
 $WP(c, B) \Rightarrow VC(c, B)$
- Try it !!

## Forward Verification Condition Generation

- Traditionally VC is computed backwards
  - Works well for structured code
- But it can also be computed in a forward direction
  - Works even for un-structured languages (e.g., assembly language)
  - Uses symbolic evaluation, a technique that has broad applications in program analysis
    - e.g. the PREfix tool (Intrinsa, Microsoft) works this way

## Symbolic Evaluation

- Consider the language of instructions:  
 $x := e \mid f() \mid \text{if } e \text{ goto } L \mid \text{goto } L \mid L : \mid \text{return} \mid \text{inv } e$
- The "inv  $e$ " instruction is an annotation
  - Says that boolean expression  $e$  holds at that point
- Programs are sequences of instructions
- Notation:  $I_k$  is the instruction at address  $k$

## Symbolic Evaluation. Basic Intuition

- VC generation is traditionally backwards due to assignments  
 $VC(x_1 := e_1; \dots; x_n := e_n, P) =$   
 $(P[e_n/x_n]) [e_{n-1}/x_{n-1}] \dots [e_1/x_1]$
- We can use the following rule  
 $(P[e_2/x_2])[e_1/x_1] = P[e_2[e_1/x_1]/x_2, e_1/x_1]$
- Symbolic evaluation computes the substitution in a forward direction, and applies it when it reaches the postcondition

## Symbolic Evaluation. The State.

- We set up a symbolic evaluation state:

$\Sigma : \text{Var} \rightarrow \text{SymbolicExpressions}$

$\Sigma(x)$  = the symbolic value of  $x$  in state  $\Sigma$   
 $\Sigma[x:=e]$  = a new state in which  $x$ 's value is  $e$

We shall use states also as substitutions:

$\Sigma(e)$  - obtained from  $e$  by replacing  $x$  with  $\Sigma(x)$

So far this is pretty much like the operational semantics

## Symbolic Evaluation. The Invariants.

- The symbolic evaluator keeps track of the encountered invariants
- A new element of execution state:  $\text{Inv} \subseteq \{1..n\}$
- If  $k \in \text{Inv}$  then
  - $I_k$  is an invariant instruction that we have already executed
- Basic idea: execute an *inv* instruction only twice:
  - The first time it is encountered
  - And one more time around an *arbitrary* iteration

## Symbolic Evaluation. Rules.

- Define a VC function as an interpreter:

$VC : 1..n \times \text{SymbolicState} \times \text{InvariantState} \rightarrow \text{Assertion}$

$VC(L, \Sigma, \text{Inv})$	if $I_k = \text{goto } L$
$e \Rightarrow VC(L, \Sigma, \text{Inv}) \wedge$	
$\neg e \Rightarrow VC(k+1, \Sigma, \text{Inv})$	if $I_k = \text{if } e \text{ goto } L$
$VC(k+1, \Sigma[x := \Sigma(e)], \text{Inv})$	if $I_k = x := e$
$\Sigma(\text{Post}_{\text{current-function}})$	if $I_k = \text{return}$
$\Sigma(\text{Pre}_f) \wedge$	
$\forall a_1, \dots, a_m. \Sigma(\text{Post}_f) \Rightarrow VC(k+1, \Sigma', \text{Inv})$	if $I_k = f()$
(where $y_1, \dots, y_m$ are modified by $f$ and $a_1, \dots, a_m$ are fresh parameters and $\Sigma' = \Sigma[y_1 := a_1, \dots, y_m := a_m]$ )	

Automated Deduction - George Necula - Lecture 2

67

## Symbolic Evaluation. Invariants.

Two cases when seeing an invariant instruction:

- We see the invariant for the first time

- $I_k = \text{inv } e.$
- $k \notin \text{Inv}$
- Let  $\{y_1, \dots, y_m\}$  be the variables that could be modified on a path from the invariant back to itself
- Let  $a_1, \dots, a_m$  be fresh new symbolic parameters

$VC(k, \Sigma, \text{Inv}) =$

$\Sigma(e) \wedge \forall a_1, \dots, a_m. \Sigma'(e) \Rightarrow VC(k+1, \Sigma', \text{Inv} \cup \{k\})$

with  $\Sigma' = \Sigma[y_1 := a_1, \dots, y_m := a_m]$

(like a function call)

Automated Deduction - George Necula - Lecture 2

68

## Symbolic Evaluation. Invariants.

- We see the invariant for the second time

- $I_k = \text{inv } E$
- $k \in \text{Inv}$

$VC(k, \Sigma, \text{Inv}) = \Sigma(e)$

(like a function return)

- Some tools take a more simplistic approach
  - Do not require invariants
  - Iterate through the loop a fixed number of times
  - PREFIX, versions of ESC (Compaq SRC)
  - Sacrifice completeness for usability

Automated Deduction - George Necula - Lecture 2

69

## Symbolic Evaluation. Putting it all together

- Let

- $x_1, \dots, x_n$  be all the variables and  $a_1, \dots, a_n$  fresh parameters
- $\Sigma_0$  be the state  $[x_1 := a_1, \dots, x_n := a_n]$
- $0$  be the empty  $\text{Inv}$  set

- For all functions  $f$  in your program, compute

$\forall a_1, \dots, a_n. \Sigma_0(\text{Pre}_f) \Rightarrow VC(f_{\text{entry}}, \Sigma_0, 0)$

- If all of these predicates are valid then:

- If you start the program by invoking any  $f$  in a state that satisfies  $\text{Pre}_f$  the program will execute such that
  - At all "inv  $e$ " the  $e$  holds, and
  - If the function returns then  $\text{Post}_f$  holds
- Can be proved w.r.t. a real interpreter (operational semantics)
- Proof technique called co-induction (or, assume-guarantee)

Automated Deduction - George Necula - Lecture 2

70

## VC Generation Example

- Consider the program

Precondition:  $x \leq 0$

Loop:  $\text{inv } x \leq 6$

if  $x > 5$  goto End

$x := x + 1$

goto Loop

End: return

Postcondition:  $x = 6$

Automated Deduction - George Necula - Lecture 2

71

## VC Generation Example (cont.)

$\forall x.$

$x \leq 0 \Rightarrow$

$x \leq 6 \wedge$

$\forall x'.$

$(x' \leq 6 \Rightarrow$

$x' > 5 \Rightarrow x' = 6$

$\wedge$

$x' \leq 5 \Rightarrow x' + 1 \leq 6)$

- VC contains both **proof obligations** and assumptions about the control flow

Automated Deduction - George Necula - Lecture 2

72

## VC Generation Example (with memory)

- Consider the program
 

```

1: I := 0      Precondition: B : bool ∧ A : array(bool, L)
   R := B
3: inv I ≥ 0 ∧ R : bool
   if I ≥ L goto 9
   assert saferd(A + I)
   T := *(A + I)
   I := I + 1
   R := T
   goto 3
9: return R      Postcondition: R : bool
      
```

Automated Deduction - George Necula - Lecture 2

73

## VC Generation Example (cont.)

- $$\forall A. \forall B. \forall L. \forall \mu$$
- $$B : \text{bool} \wedge A : \text{array}(\text{bool}, L) \Rightarrow$$
- $$0 \geq 0 \wedge B : \text{bool} \wedge$$
- $$\forall I. \forall R.$$
- $$I \geq 0 \wedge R : \text{bool} \Rightarrow$$
- $$I \geq L \Rightarrow R : \text{bool}$$
- $$\wedge$$
- $$I < L \Rightarrow \text{saferd}(A + I) \wedge$$
- $$I + 1 \geq 0 \wedge$$
- $$\text{sel}(\mu, A + I) : \text{bool}$$
- VC contains both **proof obligations** and **assumptions** about the control flow

Automated Deduction - George Necula - Lecture 2

74

## VC and Invariants

- Consider the Hoare triple:
 
$$\{x \leq 0\} \text{while}_x x \leq 5 \text{ do } x := x + 1 \{x = 6\}$$
- The VC for this is:
 
$$x \leq 0 \Rightarrow I(x) \wedge \forall x. (I(x) \Rightarrow (x > 5 \Rightarrow x = 6 \wedge x \leq 5 \Rightarrow I(x+1)))$$
- Requirements on the invariant:
  - Holds on entry  $\forall x. x \leq 0 \Rightarrow I(x)$
  - Preserved by the body  $\forall x. I(x) \wedge x \leq 5 \Rightarrow I(x+1)$
  - Useful  $\forall x. I(x) \wedge x > 5 \Rightarrow x = 6$
- Check that  $I(x) = x \leq 6$  satisfies all constraints

Automated Deduction - George Necula - Lecture 2

75

## VC Can Be Large

- Consider the sequence of conditionals
 
$$(\text{if } x < 0 \text{ then } x := -x); (\text{if } x \leq 3 \text{ then } x += 3)$$
  - With the postcondition  $P(x)$
- The VC is
 
$$x < 0 \wedge -x \leq 3 \Rightarrow P(-x + 3) \wedge$$

$$x < 0 \wedge -x > 3 \Rightarrow P(-x) \wedge$$

$$x \geq 0 \wedge x \leq 3 \Rightarrow P(x + 3) \wedge$$

$$x \geq 0 \wedge x > 3 \Rightarrow P(x)$$
- There is one conjunct for each path
  - $\Rightarrow$  exponential number of paths!
  - Conjuncts for non-feasible paths have un-satisfiable guard!
- Try with  $P(x) = x \geq 3$

Automated Deduction - George Necula - Lecture 2

76

## VC Can Be Large (2)

- VCs are exponential in the size of the source because they attempt relative completeness:
  - To handle the case then the correctness of the program must be argued independently for each path
- Remark:
  - It is unlikely that the programmer could write a program by considering an exponential number of cases
  - But possible. Any examples?
- Solutions:
  - Allow invariants even in straight-line code
  - Thus do not consider all paths independently!

Automated Deduction - George Necula - Lecture 2

77

## Invariants in Straight-Line Code

- Purpose: modularize the verification task
- Add the command "after c establish I"
  - Same semantics as c (I is only for verification purposes)
  - $VC(\text{after } c \text{ establish } I, P) =_{\text{def}} VC(c, I) \wedge \forall x_i. I \Rightarrow P$
  - where  $x_i$  are the **ModifiedVars**(c)
- Use when c contains many paths
 
$$\text{after if } x < 0 \text{ then } x := -x \text{ establish } x \geq 0;$$

$$\text{if } x \leq 3 \text{ then } x += 3 \{ P(x) \}$$
- VC now is (for  $P(x) = x \geq 3$ )
 
$$(x < 0 \Rightarrow -x \geq 0) \wedge (x \geq 0 \Rightarrow x \geq 0) \wedge$$

$$\forall x. x \geq 0 \Rightarrow (x \leq 3 \Rightarrow P(x+3)) \wedge x > 3 \Rightarrow P(x)$$

Automated Deduction - George Necula - Lecture 2

78

## Dropping Paths

- In absence of annotations drop some paths
- $VC(\text{if } E \text{ then } c_1 \text{ else } c_2, P) = \text{choose one of}$ 
  - $E \Rightarrow VC(c_1, P) \wedge \neg E \Rightarrow VC(c_2, P)$
  - $E \Rightarrow VC(c_1, P)$
  - $\neg E \Rightarrow VC(c_2, P)$
- We sacrifice soundness!
  - No more guarantees but possibly still a good debugging aid
- Remarks:
  - A recent trend is to sacrifice soundness to increase usability
  - The PREFIX tool considers only 50 non-cyclic paths through a function (almost at random)

Automated Deduction - George Necula - Lecture 2

79

## VCGen for Exceptions

- We extend the source language with exceptions without arguments:
  - `throw` throws an exception
  - `try c1 handle c2` executes `c2` if `c1` throws
- Problem:
  - We have non-local transfer of control
  - What is  $VC(\text{throw}, P)$ ?
- Solution: use 2 postconditions
  - One for normal termination
  - One for exceptional termination

Automated Deduction - George Necula - Lecture 2

80

## VCGen for Exceptions (2)

- Define:  $VC(c, P, Q)$  is a precondition that makes `c` either not terminate, or terminate normally with `P` or throw an exception with `Q`
- Rules
  - $VC(\text{skip}, P, Q) = P$
  - $VC(c_1; c_2, P, Q) = VC(c_1, VC(c_2, P, Q), Q)$
  - $VC(\text{throw}, P, Q) = Q$
  - $VC(\text{try } c_1 \text{ handle } c_2, P, Q) = VC(c_1, P, VC(c_2, Q, Q))$
  - $VC(\text{try } c_1 \text{ finally } c_2, P, Q) = ?$

Automated Deduction - George Necula - Lecture 2

81

## VCGen for Exceptions (3)

- What if we have exceptions with arguments
- Introduce global variable `ex` for the exception argument
- The exceptional postcondition can now refer to `ex`
- Remember that we must add an exceptional postcondition to functions also
  - Like the `THROWS` clause in Java or `Modula-3`

Automated Deduction - George Necula - Lecture 2

82

## Additional Language Extensions

- `wrong`
  - Semantics: terminate execution with an error
  - $VC(\text{wrong}, P) = \text{false}$
- `fail`
  - Semantics: terminate execution with a benign error
  - Used to denote error conditions that are allowed (e.g. index out of bounds)
  - $VC(\text{fail}, P) = \text{true}$
- `assert E = if E then skip else wrong`
  - Used to specify verification goals
- `assume E = if E then skip else fail`
  - Used to specify assumptions (e.g. about inputs)

Automated Deduction - George Necula - Lecture 2

83

## Additional Language Extensions (2)

- Undefined variables
  - $VC(\text{let } x \text{ in } c, P) = \forall x. VC(c, P)$

Automated Deduction - George Necula - Lecture 2

84

## Call-by-Reference

- Let  $f(\text{VAR } x, y) \{ x \leftarrow 5; f \leftarrow x + y \}$
- with
  - $\text{Pre}_f = y \geq 10$
  - $\text{Post}_f = f \geq 15$
- VC of the body is  $\forall xy. y \geq 10 \Rightarrow 5 + y \geq 15$ 
  - Which is provable!
- Let  $\{x = 10\} y \leftarrow f(x, x) \{y \geq 15\}$
- Its VC is  $x = 10 \Rightarrow (x \geq 10 \wedge (\forall y. y \geq 15 \Rightarrow y \geq 15))$ 
  - Also provable
  - But if we run the code, y is 10 at the end!
  - What's going on?

## Call-by-Reference

- Axiomatic semantics can model only call-by-value
- Not a big problem, just have to model call-by-reference with call-by-value
  - Pass pointers instead
  - Use the sel/upd axioms instead of assignment axioms

## Arrays

- Arrays can be modeled like we did pointers
- In a safe language (no & and no pointer arithmetic)
  - We can have a store value for each array
  - Instead of a unique store  $\mu$
- $A[E]$  is considered as  $\text{sel}(A, E)$
- $A[E_1] \leftarrow E_2$  is considered as  $A \leftarrow \text{upd}(A, E_1, E_2)$
- What is the advantage over  $\text{sel}(\mu, A+E)$ ?

## Mutable Records - Two Models

- Let  $r : \text{RECORD } f1 : T1; f2 : T2 \text{ END}$
- Records are reference types
- Method 1
  - One "memory" for each record
  - One index constant for each field. We postulate  $f1 \neq f2$
  - $r.f1$  is  $\text{sel}(r, f1)$  and  $r.f1 := E$  is  $r := \text{upd}(r, f1, E)$
- Method 2
  - One "memory" for each field
  - The record address is the index
  - $r.f1$  is  $\text{sel}(f1, r)$  and  $r.f1 := E$  is  $f1 := \text{upd}(f1, r, E)$

## VC Generation for Assembly Language

- Issues when extending this to real assembly language
- A fixed set of variables
  - Machine registers
- Handling of stack slots
  - Option 1: as memory locations (too inefficient)
  - Option 2: as additional registers (reverse spilling)
    - Careful with renaming at function calls and returns
    - Does not work in presence of address-of operator
- Two's complement arithmetic
  - Careful to distinguish between + and  $\oplus$
- See Necula-Ph.D. thesis for x86 and Alpha example

## VC as a "Semantic CheckSum"

- Weakest preconditions are an expression of the program semantics
  - Two equivalent programs have (logically) equivalent WP
- VC are almost the same
  - Except for the loop invariants and function specifications
- In fact, VC abstract over some syntactic details
  - Order of unrelated expressions
  - Names of variables
  - Common subexpressions

## VC for Type Checking Low Level Code

- Type checking low level code is hard because
  - We cannot have variable declarations
    - Each instance of a register use can have different types
  - The type of a register depends on the data flow
    - Varies depending on how a use was reached
- Consider the program and its VC

```

x ← 4           4 == 5 : bool ∧ ¬ (4 == 5)
x ← x == 5
assert x : bool
x ← not x      • No live logical variables
assert x       • Not confused by reuse of x
    
```

Automated Deduction - George Necula - Lecture 2

91

## Invariance of VC Through Optimizations

- In fact, VC is so good at abstracting syntactic details that it is invariant through many common optimiz.
  - Preserves syntactic form, not just semantic meaning!
- We'll take a look at some optimizations and see how VC changes/doesn't change
- This can be used to verify correctness of compiler optimizations (Translation Validation)

Automated Deduction - George Necula - Lecture 2

92

## Dead-Code Elimination

- VC is insensitive to dead-code
- The symbolic evaluator won't even look at it!

Before		After	
Code	VC	Code	VC
1 $i = e_1$			
2 <b>go to</b> L			
3 $i = e_2$	$[e_1/i]P$	$i = e_1$	$[e_1/i]P$
4 L: ...P...		L: ...P...	

Automated Deduction - George Necula - Lecture 2

93

## Common-Subexpression Elimination

Before		After	
Code	VC	Code	VC
1 $i = e_1$		$i = e_1$	
2 $j = e_1$	$[e_1/i, e_1/j]P$	$j = i$	$[e_1/i, e_1/j]P$
3 ...P...		...P...	

- CSE does not change the VC

Automated Deduction - George Necula - Lecture 2

94

## Copy Propagation

Before		After	
Code	VC	Code	VC
1 $i = e_1$		$i = e_1$	
2 $j = i$	$[e_1/i, e_1/j]P$	... $[j/j]P$ ...	$[e_1/i, e_1/j]P$
3 ...P...			

- Does not change the VC
- Just like CSE

Automated Deduction - George Necula - Lecture 2

95

## Instruction Scheduling

Before		After	
Code	VC	Code	VC
1 $i = e_1$		$j = e_2$	
2 $j = e_2$	$[e_1/i, e_2/j]P$	$i = e_1$	$[e_1/i, e_2/j]P$
3 ...P...		...P...	

- Instruction scheduling does not change VC

Automated Deduction - George Necula - Lecture 2

96

## Register Allocation

Before		After	
Code	VC	Code	VC
$i = e$ $j = e'$ $\dots$	$\left[ \left[ \frac{e}{i} \right] \left[ \frac{e'}{j} \right] P \right]$	$r_j = \left[ \frac{e}{i} \right] e$ $r_j = \left[ \frac{e'}{i} \right] e'$ $\dots \left[ \frac{e'}{j} \right] P \dots$	$\left[ \frac{e}{i} \right] \left( \left[ \frac{e'}{i} \right] \left[ \frac{e'}{j} \right] P \right)$

- Does not change VC
  - Final logical variables are quantified over
- Even spilling can be accommodated
  - By giving register names to spill slots on the stack

Automated Deduction - George Necula - Lecture 2

97

## Loop Invariant Hoisting

Before		After	
Code	VC	Code	VC
$L_1: \text{if } C \text{ go to } L_2$ $i = e_1$ $\dots P \wedge I \dots$ $L_2: \dots P' \dots$	$I \wedge$ $\forall i. I \supset ((C \supset P') \wedge$ $(-C \supset \left[ \frac{e}{i} \right] I) \wedge$ $(-C \supset \left[ \frac{e'}{i} \right] P))$	$j = e_1$ $L_1: \text{if } C \text{ go to } L_2$ $i = j$ $\dots P \wedge I \dots$ $L_2: \dots P' \dots$	$I \wedge$ $\forall i. I \supset ((C \supset P') \wedge$ $(-C \supset \left[ \frac{e}{j} \right] I) \wedge$ $(-C \supset \left[ \frac{e'}{j} \right] P))$

- VC is not changed
- For same reasons as with CSE

Automated Deduction - George Necula - Lecture 2

98

## Redundant Conditional Elimination

Before		After	
Code	VC	Code	VC
$\text{if } \neg C_1 \text{ go to } L_2$ $\text{if } \neg C_2 \text{ go to } L_2$ $\dots$ $\text{if } \neg C_n \text{ go to } L_2$ $\text{if } C \text{ go to } L_1$ $\dots P' \dots$ $L_1: \dots P' \dots$ $L_2: \dots P \dots$	$\neg C_1 \supset P \wedge$ $C_1 \supset (\neg C_2 \supset P) \wedge$ $(C_2 \supset \dots$ $C_n \supset (C \supset P') \wedge$ $(\neg C \supset P'))$	$\text{if } \neg C_1 \text{ go to } L_2$ $\text{if } \neg C_2 \text{ go to } L_2$ $\dots$ $\text{if } \neg C_n \text{ go to } L_2$ $\text{inv } C, \{ \}$ $\dots P' \dots$ $L_2: \dots P \dots$	$\neg C_1 \supset P \wedge$ $C_1 \supset (\neg C_2 \supset P) \wedge$ $(C_2 \supset \dots$ $C_n \supset C \wedge$ $C \supset P')$

- VC changes syntactically
- But not semantically (since  $C_1 \wedge C_2 \wedge \dots \wedge C_n \Rightarrow C$ )
  - We do have to prove something here
- Same for array-bounds checking elimination

Automated Deduction - George Necula - Lecture 2

99

## VC Characterize a Safe Interpreter

- Consider a fictitious "safe" interpreter
  - As it goes along it performs checks (e.g. safedr, validString)
  - Some of these would actually be hard to implement
- The VC describes all of the checks to be performed
  - Along with their context (assumptions from conditionals)
  - Invariants and pre/postconditions are used to obtain a finite expression (through induction)
- VC is valid ) interpreter never fails
  - We enforce same level of "correctness"
  - But better (static + more powerful checks)

Automated Deduction - George Necula - Lecture 2

100

## Conclusion

- Verification conditions
  - Are an expression of semantics and specifications
  - Completely independent of a language
  - Can be computed backward/forward on structured/unstructured code
  - Can be computed on high-level/low-level code
- Using symbolic evaluation we can hope to check correctness of compiler optimizations
  - See "Translation Validation for an Optimizing Compiler" off the class web page
- Next: We start proving VC predicates

Automated Deduction - George Necula - Lecture 2

101